

**Second Revised submission for
MOF Model to Text
Transformation Language
RFP**

(OMG Document: ad/04-04-07)

MOFScript

Second Revised submission

OMG document ad/2005-11-03

Version 0.3 (2005/11/14)

Submitted by:

Softeam

Supported by:

**SINTEF, LIP6, Univ. York, Thales, UPM, France Telecom, Fraunhofer
FOKUS**

Copyright ©2005 Softeam

Copyright ©2005 SINTEF

The companies and individuals listed above hereby grants a royalty-free license to the Object Management Group, Inc. (OMG) for worldwide distribution of this document or any derivative works thereof within OMG and to OMG members for evaluation purposes, so long as the OMG reproduces the copyright notices and the below paragraphs on all distributed copies.

The companies and individuals listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version.

The copyright holders listed above have agreed that no person shall be deemed to have infringed the copyright, in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

NOTICE: The information contained in this document is subject to change with notice. The material in this document details a submission to the Object Management Group for evaluation in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification by the submitter. WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES AND INDIVIDUALS LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

The Object Management Group and the companies and individuals listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

This document contains information that is patented which is protected by copyright. All Rights Reserved. No part of the work covered by copyright hereon may be reproduced or used in any form or by any means -- graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems -- without permission of the copyright owners. All copies of this document must include the copyright and other information contained on this page.

The copyright owners grant member companies of the OMG permission to make a limited number of copies of this document (up to fifty copies) for their internal use as part of the OMG evaluation process. RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical, Data and Computer Software Clause at DFARS 252.227.7013.

OMG is a registered trademark of the Object Management Group, Inc.

Contents

CHAPTER I - Preface.....	5
CHAPTER II - Executive Summary	7
CHAPTER III - Part I – Response to the RFP	10
1. Introduction.....	11
1.1 An overview of the RFP.....	11
1.2 Resolution of RFP requirements	11
1.2.1 Mandatory requirements	11
1.2.2 Optional requirements.....	13
1.2.3 Issues to be discussed	13
1.2.4 Relationship to Existing OMG Specifications	13
CHAPTER IV - Part II – Technical Part.....	14
2. Model 2 Text - MOFScript specification.....	15
2.1 Overview.....	15
2.2 Text Transformation	16
2.2.1 TextTransformation	17
2.2.2 The TextMappingDescriptor.....	17
2.2.3 LibraryImport	18
2.3 TextMappingOperations	18
2.3.1 TextMappingOperation.....	19
2.3.2 Entry point rule	19
2.3.3 TextOperationBody	20
2.3.4 TextMappingSection.....	20
2.4 Variables and constants.....	20
2.5 Expressions	21
2.5.1 Output expressions.....	21
2.5.2 Iterators.....	24
2.5.3 ForEachExp	24
2.5.4 IfExp	25
2.6 Extensibility	25
2.7 Types.....	25
2.7.1 String type.....	25
2.7.2 Integer type.....	27
2.7.3 Real type	27
2.7.4 Boolean type	27
2.7.5 Dictionary type	28
2.7.6 List type	28
2.7.7 Tuple type	29
2.7.8 XML library	29

2.7.9	Utility library	30
2.7.10	Traceability and protected regions	32
2.8	Honouring Hand made changes	32
2.8.1	Trace metamodel	32
2.8.2	Protecting regions	35
2.8.3	Relationship to QVT Traceability Model.....	36
2.8.4	Changes to source models.....	36
2.8.5	Reverse engineering.....	36
2.8.6	Round-trip engineering	36
2.9	Textual syntax.....	37
2.9.1	Text transformation.....	37
2.9.2	Importing transformations	37
2.9.3	Entry point operations (rules)	37
2.9.4	Text Mapping Operations (Rules).....	37
2.9.5	Files	38
2.9.6	Printing output	39
2.9.7	Escaped output.....	39
2.9.8	Comments.....	40
2.9.9	Properties.....	40
2.9.10	Variables.....	40
2.9.11	Using collections	40
2.9.12	Iterators.....	41
2.9.13	If-then-else.....	42
2.9.14	Invoking rules	42
2.9.15	Return results from rules.....	42
2.9.16	Grammar.....	42
2.10	Detailed Semantics.....	44
2.10.1	Files and devices.....	44
2.10.2	Print statements.....	44
3.	Compliance points.....	45
	CHAPTER V - Part III - Appendices.....	46
4.	Appendix A Examples	46
3.1	UML 2 Java	46
3.2	UML 2 WSDL	49
3.3	UML to HTML	54
	CHAPTER VI - References.....	57

Preface

This version of the submission

This is the second revised submission to the MOF model to text Transformation RFP

Submission contact points

Jon Oldevik, jon.oldevik at sintef.no

Philippe Desfray, philippe.desfray at softeam.com

Guide to the material in the submission

This document is structured in 4 parts:

- The executive summary part, which gives the highlights of the submission.
- Part I, which introduces RFP requirements and their respective resolving in this proposal.
- Part II, the technical part, which describes the contribution in terms of concepts/metamodels, syntax and semantics.
- Part III, which contains additional examples.

Statement of proof of concept

A prototype implementation of the MOFScript language is currently being developed by SINTEF through work in the European IP project 511731 MODELWARE (<http://www.modelware-ist.org>).

The prototype is available as an Eclipse plugin, available for download at <http://www.modelbased.net/mofscript>.

Resolution of RFP requirements and requests

See Section 1.2.

Submitters

This revised submission is submitted by the following OMG members:

- Softeam

Supporters

This document is supported by the following OMG members:

- France Telecom
- Fraunhofer FOKUS
- SINTEF
- Thales
- University of Paris IV (LIP6),
- Universidad Politécnica de Madrid (UPM)
- University of York

Submission team

The following people have been involved in this version of the submission:

SINTEF: Jon Oldevik, Tor Neple, Roy Grønmo, Jan Aagedal

Softeam: Philippe Desfray

Version history

Second revised submission, document version 0.3: This is the third submission submitted to the OMG.

Revised submission, document version 0.2: Second and revised version submitted to the OMG.

Initial submission, document version 0.1, OMG document number ad/2005-01-02: First version submitted to the OMG.

Future direction

The future direction of this proposal is focus on QVT-compatibility and reuse, usability, and management of manual changes.

Executive Summary

This submission presents the second revised Softeam/SINTEF submission for the MOF model to text Transformation standard called MOFScript. This section gives a brief overview of its key concepts.

Model to text transformations are tailored transformations that produce text output from models based on metamodels. This specification aims for alignment with the QVT specification, which is done by means of specializing the QVT metamodel concepts using the current QVT-Merge submission [2] as reference.

The main aim of the MOFScript is to provide a set of facilities that provide a high usability factor for end users writing text transformations from models, including text manipulation utilities and specialised libraries for XML output.

QVT Extensions

The MOFScript language is designed with the aim of reusing QVT with a limited set of new concepts. Many concepts of QVT will not be essential for model to text transformations. On the other side, there are additional requirements in model to text transformations that are not covered by QVT, such as file outputting.

This submission extends the QVT-Merge submission and provides a set of extensions that provides text output capabilities from MOF models. More specifically, MOFScript extends concepts from the *QVTMappingOperations* package.

Files

In model to text transformations, we need to generate text output to file streams. Therefore, we have introduced the File concept in this language. A File is an abstract concept that may point to a concrete file stream at a named location.

Output of text

We need high-usability features to output text to output streams. This is provided in terms of escaped output, which provides a convenient way of mixing output text with model expressions, and in terms of standard (explicit) print statements (*print / println*).

Output of XML text

Since numerous text formats use XML notation, we have provided built-in support for printing XML. This support is more simplified than traditional XML libraries, yet powerful enough to express what is needed for generating XML in a user-friendly manner.

Guards

Guards are used in the same manner as defined in QVT, i.e. to guard the invocation criteria for a model to text transformation rule. Guards are defined in terms of Logical expressions that are used as conditions for a rule.

Expressions

Expressions are specialisations of QVT Expressions. MOFScript provides some extensions to expressions in order to handle File streaming, output printing, etc.

Rules

MOFScript rules are specializations of QVT operational rules, more specifically QVT MappingOperation. A MOFScript rule may or may not return a model or base type and takes any number of parameters.

A MOFScript rule may – similar to QVT Mapping Operation rules – also have a context type.

MOFScript rules are invoked explicitly. The order of rule execution is therefore given by the MOFScript code. As with QVT operational mappings, an entry point defines the starting point of execution.

String handling

Strings and string manipulations are essential in order to provide sensible text output. A number of String functions are therefore predefined in the language; *size*, *substring*, *toLower*, *toUpper*, *firstUpper*, *indexOf* etc.

Some of these operations already exist in OCL.

Variables and constants

MOFScript allows both variables and constants to be declared and used in expressions, either globally or locally in a rule. A variable's value can be manipulated during the lifetime of a transformation execution.

Variables can have a type, which is either a base type (String, Integer, Boolean, Real) or a Collection type (Dictionary/Hashtable, List)

Side effects

MOFScript has side effects, in the sense that variables can be declared and manipulated during the lifetime of a transformation.

Output descriptors / Configuration parameters

In order to provide model to text transformation tools with the capability of setting certain useful execution-time properties, MOFScript uses descriptors. It defines properties for a transformation. Some properties are standard, in the sense that they are useful for all transformations, others are specialised for a specific technology, tool or platform. An example of a descriptor property is *root output directory*.

Imports

A transformation can import and use rules defined within external transformation specifications. This is similar to QVT imports and allows for a flexible structuring of text transformations.

Honouring Hand Made Changes

Honouring hand made changes is an essential issue, which is handled in terms of a metamodel combining traceability support and protected areas of generated text.



Part I – Response to the RFP



1. Introduction

This document is the Softeam/SINTEF second revised submission to the MOF model to text Transformation Request For Proposal (RFP) document [1] issued by the Object Management Group (OMG) in April 2004.

In this chapter we aim to give an overview of our interpretation of the RFP, and an overview of our proposal. Part II of this document (Technical Part) contains a more detailed technical specification.

1.1 AN OVERVIEW OF THE RFP

The Model to Text Transformations RFP[1] asks for proposals that define a language that provides capabilities for generating text output from MOF models, e.g. in order to support code generation, documentation, etc. Important in this regard, is capabilities to perform string manipulation.

The RFP requires compatibility between model to text transformations and the forthcoming QVT standard.

1.2 RESOLUTION OF RFP REQUIREMENTS

1.2.1 Mandatory requirements

Proposals shall define a solution for generating text from MOF 2.0 models. Both a concrete syntax and a MOF metamodel shall describe the solution.

This proposal provides both a MOF 2.0 metamodel which is an extension of parts of the QVT metamodel and a concrete syntax for the solution defined in form of a BNF grammar.

Proposals shall reuse existing OMG language specifications, if applicable. Part of text generation requires a model query and navigation language that will be defined by the QVT submissions. The QVT proposal shall be the primary choice for query language conformance. If a new language is submitted, justification for not using existing languages shall be provided. It is expected that the query portion of the language should be reused. There may be extensions required for text production.

This proposal reuses the QVT specification in terms of specializing existing concepts and keeping extensions to a minimum.

Proposals shall support mapping definitions, which are defined on the same metalevel (e.g. M2) as the mapped metamodels. This implies that the queries in these mapping definitions are referring to features of classes defined using MOF. The actual transformation is occurring one meta-level lower (e.g. M1).

This proposal supports mapping definitions in terms of model to text rules, which are defined on the same level as QVT rules, i.e. the source models for a rule is a M2 model (metamodel). The actual transformations occur on M1.

Proposals shall be able to convert model data to strings to be included in the produced text.

This proposal's main purpose is to provide language mechanisms and functionality that allows model data to be used and included in produced text.

Proposals shall include functions for string manipulation. For example, some transformations will need to generate code that conforms to other coding standards, such as capitalization of method names for JavaBeans. Proposals shall reuse existing languages or interfaces for this string manipulation, if possible.

This proposal provides a set of operations for string manipulation that allows model element data to be manipulated, e.g. for capitalization of letters, replacement, trimming, etc.

Proposal shall combine clear text with model data to produce text documents. For example, a copyright notice may need to be placed within every generated implementation code file. Also, target code language keywords, such as "class" in C++ and Java, can be combined with model data. The combination of clear text with model query and model to text conversion instructions define the structure of the target document.

This proposal provides support for this feature by introducing escape characters that allows any text to be piped to the textual output (see EscapedPrintExpression). In addition to this explicit print operations have been included to allow users to add text to the output through use of operations in the script (see PrintExpression).

Proposals shall be able to support complex text transformation mappings. Most transformations between PSM and text will be isomorphic, but the specification must provide flexibility to build more complex transformations. The QVT query language is expected to be able to support complex queries.

This proposal supports simple or complex transformations in the sense that complex queries that might be needed for querying input models is supported through QVT compatibility. It also supports complex transformations in terms of file generation. Multiple output files can be part of a generation process.

Proposals shall allow multiple MOF models to be input to the transformation process. For example, markings can be kept separate from application models, which will need to be combined during transformation to produce the text document. UML 2.0 extension mechanisms such as markings, stereotypes, and profiles, are already included in MOF.

This proposal supports multiple MOF metamodels (M1) to be specified as input models. Implementations should allow for several models (M2) to be used as input.

1.2.2 Optional requirements

Proposals may address how the submission relates to round-trip engineering, the importation of modified code to update the models.

This proposal does not address round-trip engineering. This is considered outside the scope of this submission.

Proposals may also address the detection of hand made changes to generated text and prevention of overwriting those changes.

This proposal addresses detection of hand made changes in terms of a proposed metamodel for traceability and protection of generated text. This model can be populated and checked at transformation time and can be used to trace model elements to source locations, and to protect areas of generated text.

1.2.3 Issues to be discussed

Submissions shall discuss how the submission relates to existing model query and transformation languages.

Submissions shall discuss the ease of use, understandability, and maintainability of transformations, from the point of view of the application developer.

Submissions shall include example transformations of sufficient complexity to show the capability of the language.

1.2.4 Relationship to Existing OMG Specifications

Object Constraint Language (OCL) *The part of the language dealing with expressions and queries is compatible with OCL 2.0. In practice, an end-user can use the OCL syntax to navigate and query in a transformation definition. However, this proposal provides notational simplifications for usability reasons.*

Meta Object Facility (MOF): *The MOFScript language is based on EMOF as a basis and allows for definitions of text transformations from MOF metamodels.*

MOF Query View and Transformation (QVT): *The MOFScript language is an extension of QVT (based on the latest QVT-Merge submission).*



Part II – Technical Part



2. Model 2 Text - MOFScript specification

In this section, we give an overview of the model to text transformation language MOFScript. The main concerns in this submission are:

- String and text manipulation
- File output control
- Printing of text output, indirectly as escaped text or directly in print statements
- Compatibility with QVT
- Honouring hand made changes

2.1 OVERVIEW

The model to text transformation reuses the QVT specification (currently QVT-Merge)[1] and provides the extensions needed in order to enable model to text capabilities.

The package hierarchy in Figure 1 shows the MOFM2T package within the QVT metamodel package hierarchy. It uses the QVTMappingOperations package and specializes the mapping rules concepts.

The EMOF package represents the EMOF concepts from MOF 2.0. The other packages represent the corresponding packages as defined in the QVT specification.

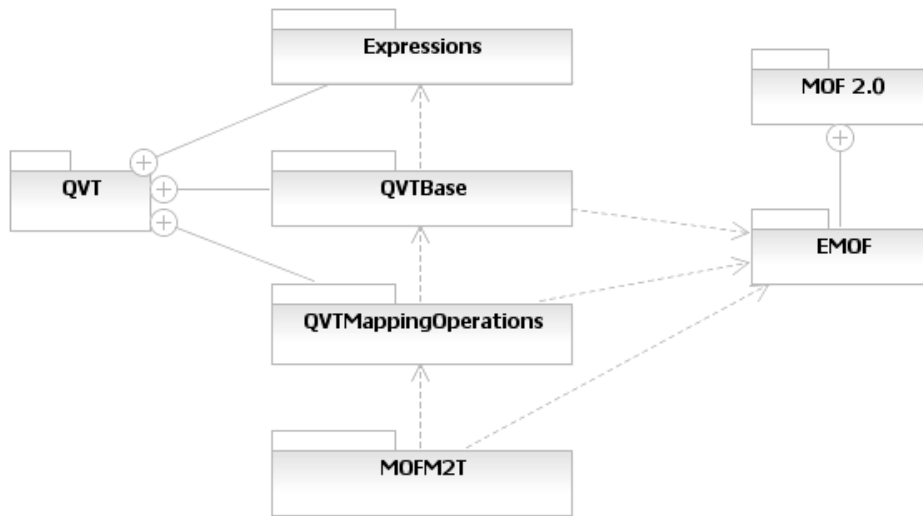


Figure 1 Meta model package overview

The MOFM2T package defines all the concepts needed in order to provide model to text transformations. These concepts are described the following sections.

2.2 TEXT TRANSFORMATION

Figure 2 shows the TextTransformation metaclass and related metaclasses. TextTransformation is the container for model to text transformation rules (operations).

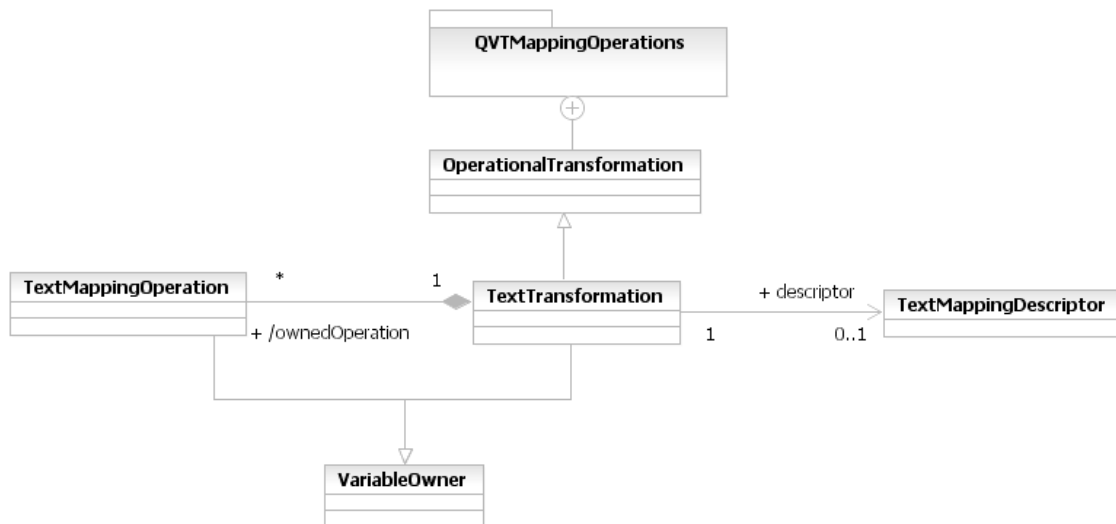


Figure 2 TextTransformation

2.2.1 TextTransformation

The TextTransformation extends the notion of OperationalTransformation from QVT-Merge. Like an OperationalTransformation, it contains a set of transformation rules. However, the transformation rules in a TextTransformation are specialised for model to text transformations.

Its properties, in addition to OperationalTransformation, are ownedTextMappings and descriptor. It also constrains some inherited properties; the usedTransformation property can only reference other text transformations.

A TextTransformation inherits from VariableOwner.

Properties:

/ownedOperation: [*] TextMappingOperation

The text mapping operations owned by a TextTransformation. It is derived from the property ownedOperation from supertype OperationalTransformation.

descriptor: [1..1] TextMappingDescriptor

The descriptor for a model to text transformation.

Constraints:

self.moduleImport.forAll(t | t.oclIsTypeOf (TextTransformation))

2.2.2 TextMappingDescriptor

The TextMappingDescriptor defines global descriptor properties for transformations. It is basically a library of configurable MOF properties that controls aspects of model to text transformations, such as directories, white space settings, etc.

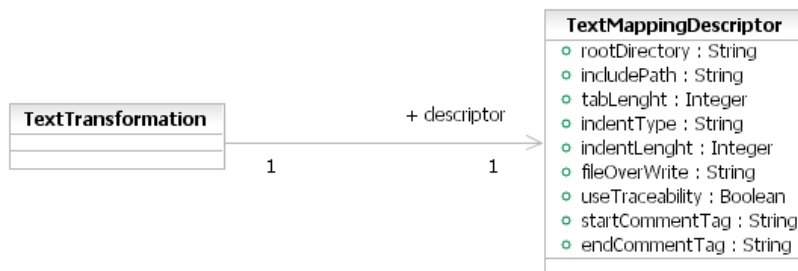


Figure 3 TextMappingDescriptor

Properties:

rootDirectory: String

The root directory advised for files produced in transformations. If this property is defined, it is used as standard for all file output devices, unless overridden specifically.

includePath: String

The include path advised for file lookup during transformations. This can hold a comma-separated list of paths or URIs.

tabLength: Integer

The length of tab characters used in a text output transformations.

indentType: String = “tab” | “space”

Controls if indent space used is default space or tab characters. If ‘space’ is specified, space characters should replace tab characters during transformation. If ‘tab’ is specified, tab characters are kept in transformations.

indentLength: Integer

Controls the length of an indent in the text, also default length for a tab.

fileOverWrite: String = “always” | “never” | “merge”

Defines the policy for file generation when a file already exists. Always overwrite, never overwrite, merge files, or backup are possible alternatives. Merge files means indicating that the transformation engine should try to merge the new file with the old file, if possible.

useTraceability: Boolean

Turns on and off traceability support.

startCommentTag

The user-defined tag which represents the start of a comment (or other ignored text in the target language). (In Java, this can be ‘/*’ or ‘//’). This is used to generate protected areas within tags in the generated text.

endCommentTag

The user-defined tag which represents the end of a comment (or other ignored text in the target language). (In Java, this can be ‘*/’ or nothing).

2.2.3 LibraryImport

The LibraryImport defines the capability for importing external text transformation. This capability is reused from the QVTMappingOperations.

2.3 TEXTMAPPINGOPERATIONS

A TextTransformation contains TextMappingOperations that define a model to text transformation. A TextMappingOperation contains a TextOperationBody. These concepts are extensions of MappingOperation concepts from QVT. TextMappingOperation extends ImperativeOperation and TextMappingBody extends OperationBody (Figure 4).

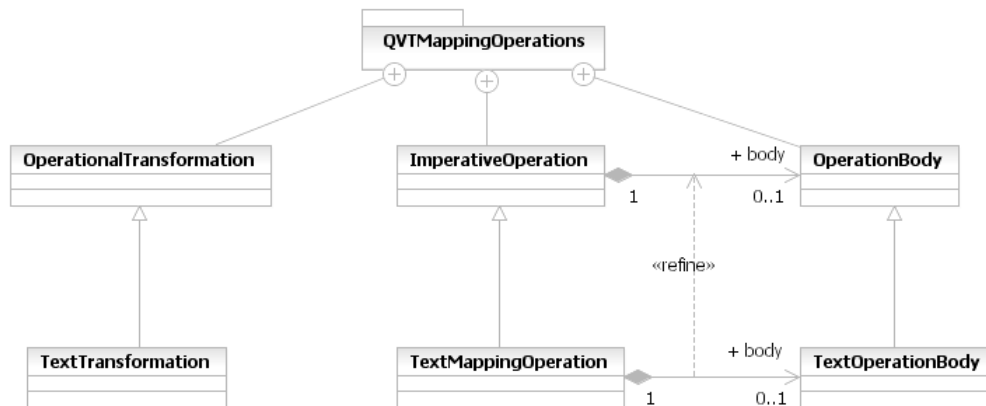


Figure 4 Text mapping operations with body

2.3.1 TextMappingOperation

A TextMappingOperation represents the concept of a transformation rule dedicated to model to text transformations, which is part of a TextTransformation. The TextMappingOperation extends ImperativeOperation from QVTMappingOperations package.

Properties:

body: [0..1] TextOperationBody

The body of a model to text transformation rule.

2.3.2 Entry point operation

An entry point rule defines execution entry points for a text transformation. It is defined by the TextMappingEntryOperation class, which inherits the EntryOperation class from QVTMappingOperations.

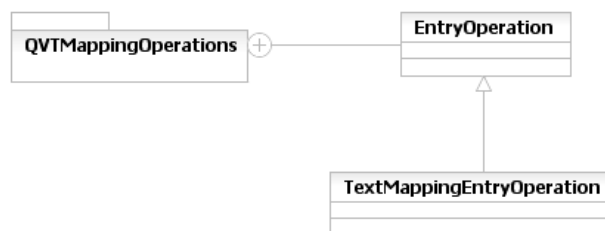


Figure 5 TextMappingEntryOperation

An entry point in a text transformation has a similar semantics as in QVT, but in addition it may be defined with a context parameter from the input metamodel.

If defined with a context parameter, it is executed once for each occurrence of the context type in the input model. If no context parameter is defined, the semantics is the same as in QVT, and the input model parameter can be used to retrieve model elements.

2.3.3 TextOperationBody

The concept `TextOperationBody` represents the body of a model to text transformation, which is part of a `TextMappingOperation`. It extends the `OperationBody` from the `QVTMappingOperations` package.

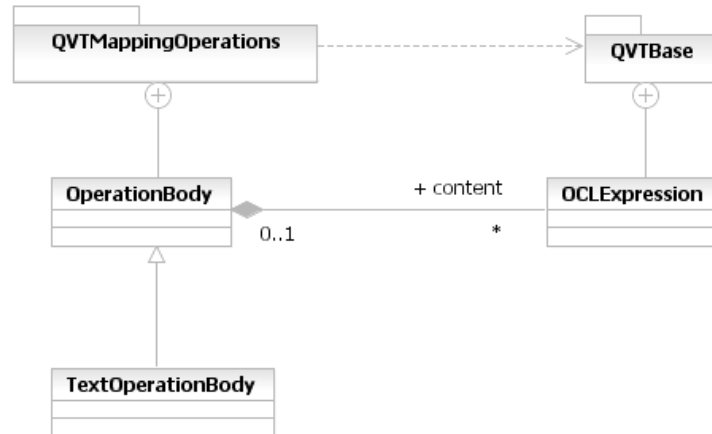


Figure 6 `TextOperationBody`

`TextOperationBody` inherits the properties of `OperationBody`. The *content* of a `TextOperationBody` is provided by specialisations of `OCLEExpression`.

2.3.4 TextMappingSection

The `M2TSection` concept represents a section within an `M2TRuleBody`, i.e. a part of a model to text transformation. It extends the `MappingSection` type from the `QVTMappingOperations` package. (*Presently, it does not introduce any new properties.*)

Properties:

2.4 VARIABLES AND CONSTANTS

`QVTMappingOperations` define variable declarations and initialisation expressions. It also defines properties, which represents local constants within a text transformation definition. These are reused as is in the text transformation specification.

Constants are defined as QVT local properties and variables as QVT variables (Figure 7). This does not represent a metamodel extension to QVT. The `TextTransformation` package only reuses concepts already defined.

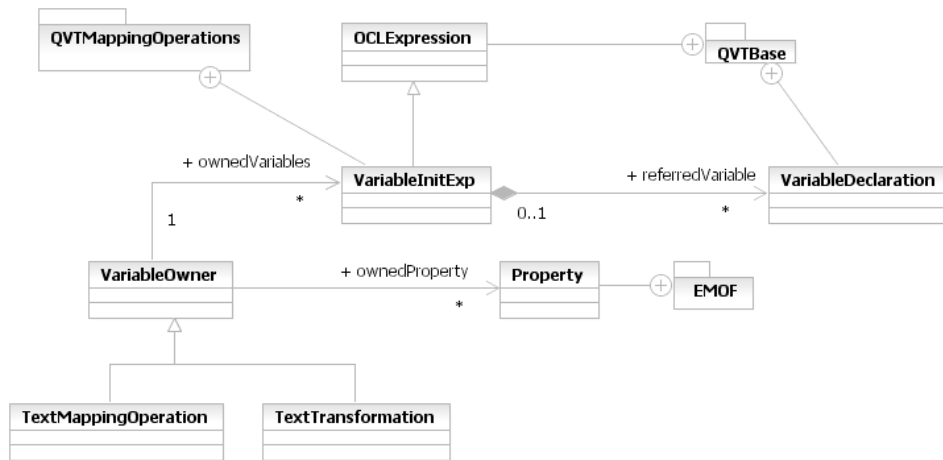


Figure 7 Variables and constants

Variables and constants can be defined globally for a transformation, or locally within a single rule. Variables are owned by a VariableOwner, either a TextTransformation or a TextMappingOperation.

2.5 EXPRESSIONS

Model to text transformations reuse the expression concept defined in QVT, and introduce some new expression types, which can be part of model to text transformation rules. The new expression meta classes are introduced in order to handle lexical output in transformations.

2.5.1 Output expressions

The output expressions control how a transformation produces textual output. Output is always produced to an output device, normally a file. It can also be the standard output stream, or perhaps other output devices. (Other output devices could be targeted, such as CVS/SVN repository streams or SQL DB streams. This will however require a metamodel extension.)

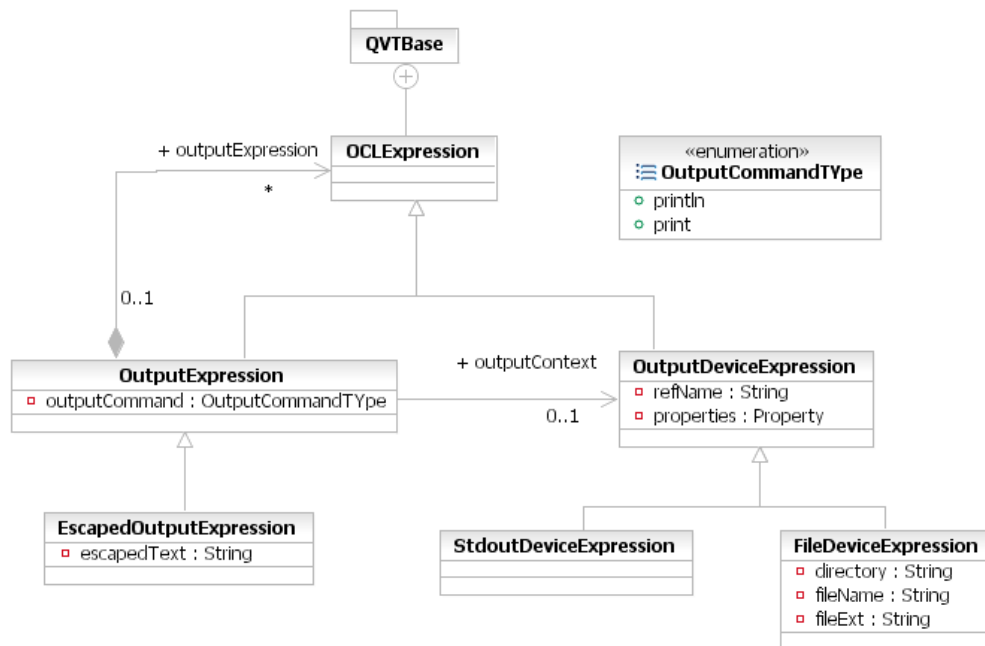


Figure 8 Output expressions

2.5.1.1 OutputExpression

OutputExpression represents an expression that generates textual output to some device, typically an output file or standard output console.

Properties:

outputBody: [1..1] Expression

The body of an OutputExpression is the subject of the output. It may be any kind expression, e.g. a text literal, rule/functional

outputContext: OutputDeviceExpression

The context to which an output expression targets its output, which may be a file, standard output, or other kinds of devices.

outputCommand: String

The print command associated with the output expression; which is either print or println.

2.5.1.2 EscapedOutputExpression

An EscapedOutputExpression represents a special kind of output expression, which allows for escaped text to be part of a textual output. Escaped text is text embedded in a rule as standard text, which is sent to output as provided. This is the same mechanism used by most scripting languages for combining text output with language expressions.

Escaped output is signalled by escape characters, beginning and ending of an escape. Basically, it is a print statement that can subsume multiple lines and be combined with all expressions that evaluate to a string. Escaped text is signalled by the characters '<%' to start an escape and '%>' to end an escape. Note that all whitespace is copied to the output device.

2.5.1.3 OutputDeviceExpression

An OutputDeviceExpression is a concept that represents the declaration of an output device, such as a file. It can be referenced implicitly or explicitly within a rule and is scoped in the same manner as a variable.

Properties:

refName : [0..1] String

A name by which this output device can be referenced. If the output device has no reference name, it can be reference implicitly or indirectly through a file name.

properties : [*] Property

A set of properties associated with an output device. This may or may not be used to control specific features of different kinds of devices.

2.5.1.4 FileDeviceExpression

A FileDeviceExpression is a specific kind of device tailored for file output. It represents a pointer to a file on a file system. A usage of this device will result in output generation on a specific file.

Properties:

directory: [0..1] String

The directory of this file. A default directory setting is used if this is not defined. Configuration properties may be used instead of this directory property.

fileName: String

The filename for this file.

fileExt: String

The file extension for this file.

2.5.1.5 StdOutDeviceExpression

A StdOutDeviceExpression is a specific kind of device for standard (console) output. It has no additional properties.

2.5.2 Iterators

QVT defines a range of iterators and block expressions, and reuses (redefines) the concepts defined in OCL. The basis of iterators in OCL and QVT is the LoopExp type, which has a set of iterators and a body. IteratorExp and IterateExp represent the fundamental semantics for iterators, and represents predefined iterators such as collect, select, forAll.

A separate set of concepts are defined in QVT to support block expressions that define expressions which are executed in sequence. This submission extends the block expressions with a new expression type – ForEachExp – which basically combines a forAll expression with an execution block (This forEach expression redefines the forEach expression from QVTMappingOperations). In addition, MOFScript reuses the QVT IfExp expression.

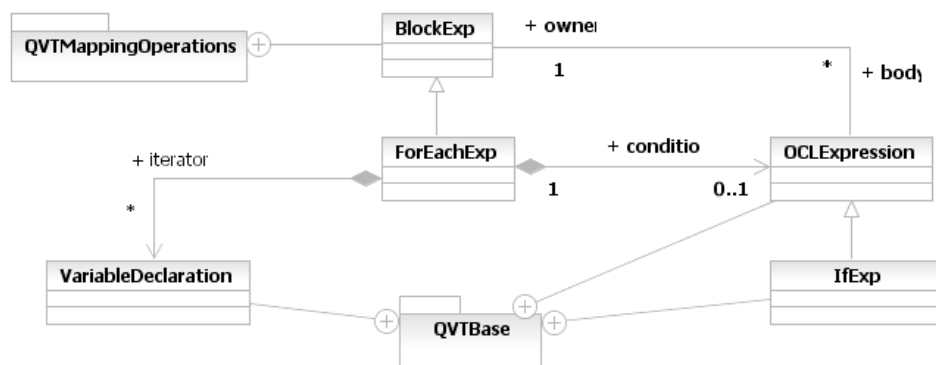


Figure 9 ForEach and If expressions

2.5.3 ForEachExp

A ForEachExp is a block expression which iterates over a collection of elements, similar to a forAll iterator, except it is followed by a block of expressions that are executed in sequence.

Properties:

iterator : [0..*] VariableDeclaration

The iterators declared for this block iterator, which are variables in the expression. These variables are, each in its turn, bound to every element value of the *source* collection while evaluating the *body* expression.

condition : [0..1] LogicalExpression

A logical expression, which may be part of the ForEachExp expression and control the selection of elements.

Iterators may be defined for Collection from a source model element, or from List/Dictionary variables, Strings, and Integers. String iterators define loops over the character contents of a string. Integer iterators define loops based on the size of the context integer. Applied on e.g. integer '3', will produce a loop running 3 times.

2.5.4 IfExp

If expressions are reused from the QVTMappingOperations. They are defined by a single 'if'-branch, followed by a set of 'else-if'-branches, and a possible 'else'-branch. Arguments to the if/else-if-branches are Boolean expressions.

The bodies of if expressions and its else branches are statement bodies (sets of expression statements in a block).

2.6 EXTENSIBILITY

An operation can override another operation by defining the same signature. A rule defined for a particular context metaclass will override a rule with the same signature defined for a context subtype.

Operation overriding has the same semantics as defined by QVT.

2.7 TYPES

MOFScript reuses the type define in OCL and QVT. In addition, it adds to some type operations, especially so for String manipulation.

The available types are: String, Integer, Boolean, Real, List, Dict/Hashtable, and the QVT Tuple type.

2.7.1 String type

The String type supports a set of String manipulation operations important for Model to Text Transformation language. The operations that are already defined in OCL 2.0, are imported; this goes for *size*, *substring*.

Standard string function library:

- String::size () : Integer
 - o The size operation returns the length of the sequence of characters represented by the object at hand.
 - o *Synonym operation: length()*
- String::substring (lower : Integer, upper: Integer) : String
 - o This operation returns a new string that is a substring of the string at hand. The substring begins at the index specified by `lower` and extends to the character at index `upper - 1`. The first character in the string has index = 0.
 - o Parameters:
 - lower: the beginning index, inclusive
 - upper: the ending index, exclusive
- String::substringBefore (match : String) : String

- This operation returns a substring of the `String` based on the first occurrence of the *match* parameter. If a match is found, it returns the substring before, and up until the location where the first *match* string is located. If a match is not found, a copy of the original `String` is returned.
- `String::substringAfter (match : String) : String`
 - This operation returns a substring of the `String` based on the first occurrence of the *match* parameter. If a match is found, it returns the substring after the location where the first *match* string is located. If a match is not found, a copy of the original `String` is returned.
- `String::toLowerCase () : String`
 - Converts all of the characters in this string to lowercase characters.
- `String::toUpperCase () : String`
 - Converts all of the characters in this string to uppercase characters.
- `String::firstToUpper () : String`
 - Converts the first character in the string to an uppercase character.
- `String::lastToUpper () : String`
 - Converts the first character in the string to a lowercase character.
- `String::indexOf (match : String) : Integer`
 - Returns the index of the first character of the first substring if the substring provided in the parameter occurs as a substring in the object at hand. If it does not occur as a substring, the operation returns -1.
 - Parameters
 - *match* : the substring to be searched for within the string.
- `String::endsWith (match : String) : Boolean`
 - Returns `true` if the string at hand ends with the substring provided in the parameter.
 - Parameters
 - *match*: the suffix to be searched for.
- `String::startsWith (match : String) : Boolean`
 - Returns `true` if the string at hand starts with the substring provided in the parameter.
 - Parameters
 - *match*: the prefix to be searched for.
- `String::trim () : String`

- Returns a copy of the string where all trailing and leading white spaces have been removed. If there are no trailing or leading white spaces the operation returns the string.
- String::normalizeSpace() : String
 - Removes all trailing and leading white space and replaces all internal sequences of white space with a single space.
- String::replace (String m1, String m2): String
 - All occurrences of m1 in the context string are replaced by m2.
- String::equals (String match) : Boolean
 - Returns true if the value of the String matches the input String 'match', else return false
- String::equalsIgnoreCase (String match) : Boolean
 - Returns true if the value of String, when ignoring letter casing matches the input String 'match', else returns false.
- String::charAt (index: Integer): String
 - Returns the character (as a String) at position 'index' in the String
- String::isUpperCase (index : Integer) : String
 - returns true if character at position 'index' is upper case. If no index is given, first position (index=0) is used.
- String::isLowerCase (index: Integer) : String
 - Returns true if character at position 'index' is lower case. If no index is given, first position (index=0) is used.

A String can be used as the source context of an iterator (forEach) expression.

2.7.2 Integer type

The Integer type is equivalent to an OCL Integer type. An integer can be used as the source context of an iterator (forEach) expression.

2.7.3 Real type

The Real type is equivalent to an OCL Real type.

2.7.4 Boolean type

The Boolean type is equivalent to an OCL Boolean type.

2.7.5 Dictionary type

A Dictionary is an unordered set of key/value pairs, equivalent to a Hashtable in Java. This is similar to the Dictionary type in QVT, but is not parameterized.

The following operations are defined for the Dict type.

- Dictionary::put (key, value)
 - o Adds a value to a hashtable
- Dictionary::get (key) : Any
 - o Gets a value from a hashtable.
- Dictionary::clear () : void
 - o Clears all elements from the collection
- Dictionary::size () : Integer
 - o Returns the size of the hashtable.
- Dictionary::isEmpty () : Boolean
 - o Returns true if the hashtable is empty, false otherwise.
- Dictionary::keys () : List
 - o Returns the key elements contained in the hashtable.
- Dictionary::values () : List
 - o Returns the value elements of the hashtable.

2.7.6 List type

A List is an ordered list of values, equivalent to a List in Java and similar to List in QVT. The List type as defined in QVT is a parameterized type. It can, however, be used untyped.

- List::add (o: Object)
 - o Adds a value (or an object) to the list
- List::remove (o: Object)
 - o Removes a value (or an object) from the list
- List::first () : Object
 - o Returns the first element of the list
- List::last () : Object
 - o Returns the last element of the list
- List::size () : Integer
 - o Returns the size of the list.
- List::clear () : void

- Clears all elements from the collection.
- List::isEmpty () : Boolean
 - Returns true if the hashtable is empty, false otherwise.

2.7.7 Tuple type

The Tuple type is a an anonymous tuple type. It has the same semantics as defined by QVT.

2.7.8 XML library

The XML Library is motivated by the widespread usage of XML specifications. Thus it is likely that in many cases the to-be-generated text format is XML, such as BPEL4WS, GML etc. This library aims at providing a small set of built-in elements and operations that enables transformations to easily specify different XML output. Figure 10 shows an overview of a simplified XML metamodel, which is the basis for this library. Note that these concepts are simplified compared to W3C's more complex metamodel which also involves more elements. It is intended that this simplified view is sufficient to handle all the XML generation needed. The benefit of a simple metamodel is that we get a small set of elements and operations that can be easy to adopt for the transformation architect. An XML element is the main element representing any XML element which can be either the root element or a fragment sub tree within the XML document. The name attribute specifies the tag name of the XML element, the namespace attribute specifies the belonging namespace and the textContent attribute defines possible text value contained directly by the XML element. The XML element may contain any number of child sub elements, a set of declared namespaces and a set of XML attributes.

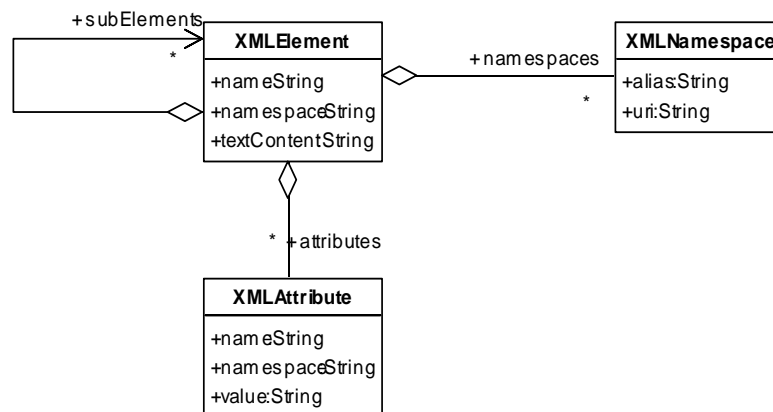


Figure 10 Logical overview of the XML metamodel

- newXMLElement (name: String) : XMLElement;
 - A new XMLElement with the given name is created. The name may contain a namespace prefix which will then be assumed to be defined within the scope when this element finally is emitted as output.
- stringToXMLElement(xmlString: String) : XMLElement

- Produces a new XML element from the given XML string
- Example: `xmlElem := stringToXMLElement("<myXMLElement name=`
`\"thisElement\">.` Here is the content`</myXMLElement>")`
- `XMLElement::toString () : String;`
 - This operation returns a string representation of the complete XML structure as represented by the `XMLElement`.
- `XMLElement::addNamespace (alias: String, uri: String) ;`
 - A new namespace declaration is added to an XML element. The namespaces added with this operation will only appear in the list of declared namespaces for this element and must not be mistaken with the namespace to which the element belongs.
- `XMLElement::setNamespace(alias: String, uri: String);`
 - This assigns the namespace to which the XML element belongs. If the alias is assigned to the empty string, then it is assumed that the namespace is used as a default namespace without explicit alias prefix.
- `XMLElement::replaceNamespace(uri: String, newAlias: String);`
 - This operation is used to easily configure the output of namespaces globally for this XML element and all its children. All namespaces belonging to the specified uri will be outputted with the newAlias as its prefix. If the newAlias is the empty string, then the namespace will be used as the default namespace without any explicit prefix.
- `XMLElement::addAttribute (attrName: String, attrValue: String);`
 - A new attribute is added to an XML element. Note: attributes will also be used for representing non-traditional attribute-like constructions with predefined meaning such as `xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance`, `xmlns="xxx"`, and `xsi:schemaLocation = "someLocation"`. Note that the attribute belongs to the same namespace as the parent element unless it has an explicit namespace prefix such as: `<a:thisElem otherNS:attrName="attrValue">`. An explicit namespace is given as part of the given input parameter `attrName` ("`otherNS:attrName`").
- `XMLElement::addElement (subElement: XMLElement)`
 - A new sub XML element is added to an XML element.
- `XMLElement::addTextContent (value: String)`
 - Declares the text content of the XML element.

2.7.9 Utility library

- `generateID () : String`

- Returns a generated id for the source context element. The id will be different for different source context elements and equal every time it is called for the same source context element.
- count () : Integer
 - Returns the count of the context of an Iterator. Calls to this function are illegal outside an iteration context and will yield a -1 value.
- position() : Integer
 - Returns the numbered “position” within an Iterator. Yields 1 on first iteration, 1 + N in subsequent N iterations. Calls to this function are illegal outside an iteration context and will yield a -1 value
- indent () : void
 - Controls indent rules in text output. Increasing the indent counter with one, forcing all proceeding output to be indented accordingly. Print functions will use the indent counter to produce indent white space.
- unindent () : void
 - Controls indent rules in text output. Decreases the indent counter with one. Proceeding output will be produced with one less indent than before.
- newline (count : Integer = 1) : void
 - Outputs a set of newline characters according to the given parameter. If no parameter is given, a single newline is produced.
- tab (count : Integer = 1) : void
 - Outputs a set of tab characters according to the given parameter. If no parameter is given, a single tab character is produced.
- space (count : Integer = 1) : void
 - Outputs a set of space characters according to the given parameter. If no parameter is given, a single space character is produced.
- time (format: String) : String
 - Time function. Returns the current time on a string format. May take a formatting string as parameter.
- date (format : String) : String
 - Date function. Returns the current date on a string format. May take a formatting string as parameter.
- getenv (envProperty : String) : String
 - Gets an environment property.
- setenv (envProperty : String, envValue: String) : void
 - Set an environment property

2.7.10 Traceability and protected regions

MOFScript introduces some keywords that allows the user to specify protected regions in the generated text.

This is based on

2.8 HONOURING HAND MADE CHANGES

Honouring hand made changes is a pertinent issue for model to text generation. Issues such as handling manual changes to generated code, changes to source models, synchronisation of generated text and the original model, traces between model information and generation code, and round-trip engineering need to be addressed in the context of MOF model to text generation.

This submission defines a trace metamodel, which allows the generation of traces from source model elements to segments in the generated text. Segments in the generated text may be part of protected blocks. Protection is provided by tags wrapped in comments in the generated text.

2.8.1 Trace metamodel

The *Trace metamodel* defines a set of concepts that enables traceability between source model elements and locations in generated text files.

Central in the trace model is the logical segmentation of a file into *blocks*, some of which may be *protected blocks*. Blocks are identifiable units within a file. At the point in time when they are generated, the blocks may also have a specific location within the file. A protected block contains a set of segments which are relatively located within the block in terms of a starting and ending offset. These segments will allow traces from source model elements to the source file. Being part of a protected block, they can also be part of a manageable file in terms of change protection.

Figure 11 shows the elements of the trace metamodel.

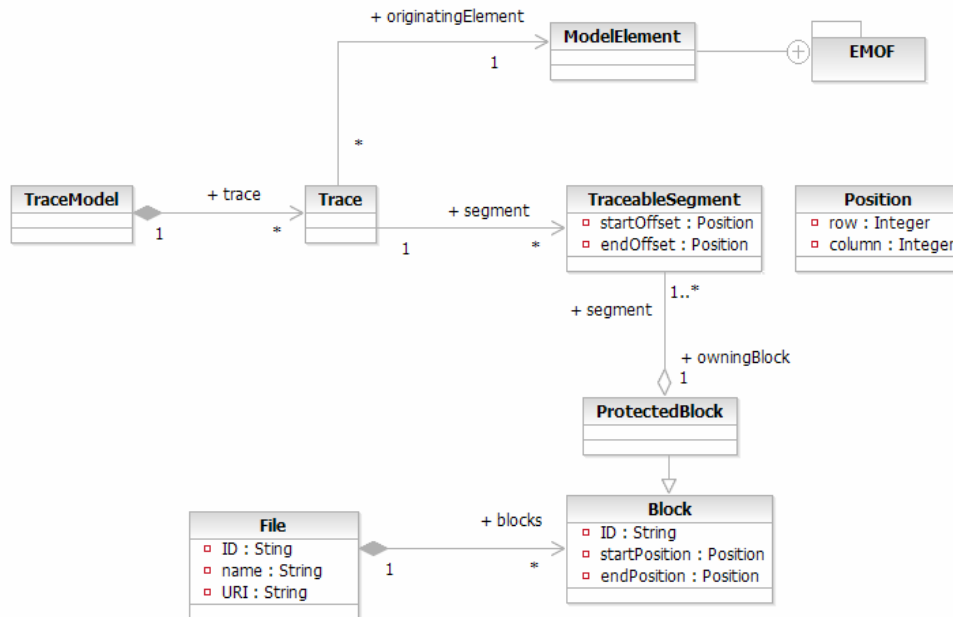


Figure 11 Trace metamodel

2.8.1.1 TraceModel

The *TraceModel* meta class is the container of a set of logically related traces. It contains a set of *Trace* instances.

Properties:

trace: [0..*] Trace

The traces owned by the trace model.

2.8.1.2 Trace

The *Trace* meta class relates a model element to a set of segments in target files.

Properties:

segment: [0..*] TraceableSegment

The segments managed by this trace.

2.8.1.3 TraceableSegment

The *TraceableSegment* meta class defines a segment within a protected block of a text file. The segment is defined in terms of a starting and ending offset, both defined in terms of a relative position to the block.

Properties:

owningBlock: ProtectedBlock

A reference to the owning block for this segment.

startOffset: Position

The starting offset of the segment, relative to the owning block. It is defined by the values of the position object by a row and column value.

endOffset: Position

The ending offset of the segment, relative to the owning block. It is defined by the values of the position object by a row and column value.

2.8.1.4 Block

A *Block* is a continuous, identifiable structure in a *File*. It can be located in a file by using its ID. It can also be located in terms of its *startPosition* and *endPosition*, when a file has not been changed.

Properties:

ID : String

A unique id for the block.

startPosition: Position

The starting position for a block.

endPosition: Position

The end position for a block.

2.8.1.5 ProtectedBlock

A *ProtectedBlock* is a block which should be protected from user changes, i.e. the user shall not enter new text into a block that is protected.

Properties:

segment: [1..*] TraceableSegment

The segments making up this protected block.

2.8.1.6 Position

A *Position* represents the position in a file, defined by a row and a column.

Properties:

row : Integer

The row of the position. May be relative to a block.

column: Integer

The column of the position.

2.8.1.7 File

A *File* represents a generated file. It contains a set of blocks.

Properties:

blocks : [0..*] Block

The blocks in a file.

ID : String

The identifier for a file

name : String

The name of a file

URI : String

The URI reference for a file.

2.8.2 Protecting regions

In order to provide protected regions for arbitrary output formats and languages, one solution is to emit protected regions in generated files using comment tags.

This is a commonly used mechanism, but it requires knowledge of the textual notation to use in each language case. This information must be provided by the user in some way.

This submission proposes a user-controlled scheme, where the control characters for ignored text (such as comment tags) must be given by the user. This is done in the two descriptor properties for a text transformation; *startCommentTag*; *endCommentTag*; which are used as delimiters when generating protected regions, together with identifying marks for the start and end of blocks. We will use the identifying marks *BOB* (*beginning of block*) and *EOB* (*end of block*)

Unprotected and protected regions will not generate themselves. If protection is enabled, the default behaviour is to protect all content as a single protected region, unless the transformation specifies otherwise through un-protecting a region.

In order to specify that a given section of a generated file is not protected, the keyword *unprotect* is used.

```
unprotect {
    <% // User can add properties %>
}
```

This will result in a splitting of the original block in two protected blocks, the first ending before the un-protected area, the other starting after the un-protected area. As an example, the following unprotect used within a Java method generation, will lead to the following code.

```
uml.Operation::operationImplementation () {
    <% public void %> self.name <% () { %>
        unprotect {
            <% /* TODO:Implement operation */ %>
        }
    }
```

```

    <%
    }
    %>
}
// Part of the generated code
// BOB blockId=123456 (start of block)
// Method within the Generated Java code
public void performAutomaticUpdates () {
    // EOB (end of block)
    /* TODO: Implement operation */
    System.out.println ("User modified code...")
    // BOB blockId=123457
}
// other code
// EOB

```

The code extract above gives a simple example of how block information can be embedded in the generated code, using the user defined properties for *startCommentTag* and *endCommentTag* (in this case, using *startCommentTag="//"*, *endCommentTag=""*).

The blocks generated are used to maintain traceability information in *TraceModel* instances.

An end user changing a generated text file can only be allowed to make changes outside protected blocks. This may be enforced by tool editors or simply by process enforcement by the end user.

2.8.3 Relationship to QVT Traceability Model

???

2.8.4 Changes to source models

Changes to source models are only an issue if the already generated text/code has been manually modified, and not yet synchronized with the model.

In this case, traceability information can be used to synchronize modified text with newly generated.

2.8.5 Reverse engineering

Reverse engineering is considered out of scope of this submission.

2.8.6 Round-trip engineering

Round-trip engineering is considered out of scope of this submission.

2.9 TEXTUAL SYNTAX

Model to text transformation is primarily concerned with textual notations, as graphical notations are less useful when expressing lexical output. MOFScript is a purely lexical notation.

The textual syntax for MOFScript is similar to that of QVT, with some exceptions to support output printing etc.

2.9.1 Text transformation

A text transformation module is denoted with the key word “transformation” followed by a name for the module. The initial part of the module is identical to QVT mapping rule module.

```
texttransformation testAnnotations (in uml:UML2)
```

2.9.2 Importing transformations

A text transformation module can import other modules and utilize rules defined there.

```
access Uml2wsdl ("uml2wsdl-lib.m2t")
```

2.9.3 Entry point operations (rules)

Entry point rules defines where the transformation starts execution. It is similar to a Java main and equivalent to QVT entry points. The syntax is the same as in QVT, with the addition of a possible context type. Some example are given below.

```
uml.Model::main () {
    self.ownedMember->forEach(p:uml.Package)
        p.transformPackage ()
}

uml.Package::main() {
    self.transformPackage()
}

main () {
    uml.objectsOfType(uml.Package)->transformPackage()
}
```

A transformation can only have a single entry point.

2.9.4 Text Mapping Operations (Rules)

Text mapping operations are defined with a name and a context type. A rule may have a return type. It may also have a guard.

The syntax is similar to QVTMappingOperation syntax. There is no specific keyword associated with a rule.

```
uml.Class::classToJava() {
```

```

    // statements
}

```

The guard for a rule is defined in the same manner as guards in QVTMappingOperations, using a 'when' clause.

```

uml.Class::classToJava ()
    when {self.getStereotype() = 'Entity'}
    {
        // statements
    }

```

A rule may also return a value, which can be reused in expressions in other rules. To return a value, the *result statement* is used.

```

uml.Package::getFullName (): String {
    if (self.owner != null)
        result = self.owner.getFullName() + "."
    else if (self.ownerPackage != null)
        result = self.ownerPackage.getFullName() + "."

    result += self.name.toLowerCase().replace(" ", "_");
}

```

A rule may have any number of parameters. A parameter can be of a built-in type or a metamodel type.

```

uml.Model::testParameters2 (s1:String, i1:Integer) {
    stdout.println("testParameters2: " + s1 + ", " + i1)}

uml.Model::testParameters3 (s1:String, r2:Real, b1:Boolean,
package:uml.Package) {
    stdout.println("Package:" + package.name)
    stdout.println ("testParameters3: " + s1 + ", " + r2 + ", " + b1 + "
" + package.name)
}

```

2.9.5 Files

Files are the most important kind of output device for text. A file is declared with a set of properties: File name, File extension, file directory, and file type. File name must always be present. File name and directory can be specified as separate properties. It may also be embedded in the file name property.

A file can be used implicitly or explicitly in output statements. For example, if a file device is declared, proceeding output statements will use that device as target. If several file devices are declared, the latest one is used by default. If a specific device is the target, it can be referenced by its name.

```

file ("an-output-file.txt")
<%
    Currently, the generator outputs a lot of gibberish to the current
    output device (file).
%>
file f2 ("AnotherFile.txt")
file ("Yet-another.txt")
println (" Now, I am writing to the file 'yet-another.txt.txt'")

```

```
f2.println (" Now, I am writing to the file 'AnotherFile.txt'");
```

2.9.6 Printing output

Output printing is done by using standard print functions or escaped output. The standard print functions are either `'print'` or `'println'`, which outputs a expression and adds a newline in the latter case.

A couple of other utility print functions are defined, to provide easier whitespace management: `newline` (or `nl`), `tab`, or `space`, followed by an optional count integer. Standard String escape characters (`\n\t`) are also legal within String literals.

```
print ("This is a standard print statement " + aVar.name)
newline (10)
tab(4) <% More escaped output \n\n %>
println (" /** Documentation output */ ");
```

A print expression (explicit) can be prefixed with a file reference (`f1.println`) or a reference to standard out (`stdout.println`). A prefix will override the current default output device.

2.9.7 Escaped output

Escaped output provides a different and in some cases simpler way of providing output to a device. Escaped output works similar to most scripting languages, such as Java script.

Escaped output is signalled by escape characters, beginning and ending of an escape. Basically, it is a **print** statement that can subsume multiple lines and be combined with all expressions that evaluate to a string. Escaped text is signalled by the characters `'<%'` to start an escape and `'%>'` to end an escape. Note that whitespace is copied to the output device.

```
uml.Operation::bindingOperation () {
  <%
    <operation name="%> self.name <%">
      <soap:operation soapAction="%> nameSpaceBase + self.name <%"
style="document"/>
      <input>
        <soap:body%>
          if (self.ownedParameter.size() > 0) {
            <% parts="%> self.getParameterOrder() <%"%>
          }
          <% use="literal"/>
        </input>
      </output>
      <soap:body%>
        if (self.returnResult.size() > 0){
          <% parts="response"%>
        }
        <% use="literal"/>
      </output>
    </operation>
  %>
}
```

2.9.8 Comments

Comments in MOFScript are denoted as in QVT, starting with ‘--’, which denotes a single line comment. In addition, single line comments in Java style ‘//’ are legal, as well as multi-line comments in Java Style ‘/*’ ‘*/’.

```
-- this is a standard QVT-like comment on a single line
// this is a standard Java-like comment on a single line
/*  this is a standard Java-like
 *   comment on
 *   several lines
 */
```

2.9.9 Properties

Properties are used in the same manner as in QVT. They can be defined at the module level or within a rule. There are two types of properties. Local properties, which are constants within a module or a rule, and configuration properties, which are defined as configuration parameters to a module, and used within many transformation specifications.

```
property JavaPackageName = "org.sintef"
```

A property cannot be modified after its declaration. It is typically used in output statements.

```
<% The Java package name is: %> JavaPackageName <% Nothing more,
nothing less %>
```

2.9.10 Variables

Variables are defined and used as in QVT. They can be defined globally for a module, or locally within a rule. A variable can have an assigned value when declared, which can be modified during its lifetime.

```
var exportCounter = 0
```

A variable can be typed. The standard OCL types are used (String, Boolean, Integer, Real). Also, the Collection types List and Hashtable are supported.

```
var modifiableName:String = "temporary name";
var storedNames:Dictionary;
```

2.9.11 Collections

Collections are for holding sets of values during transformation execution, e.g. to temporarily store pre processed information that is needed several times during generation.

```
var packageNames:List
var packageIdList:Dictionary
self.ownedMember->forEach(p:uml.Package) {
    packageNames.add (p.name)
    packageIdList.put (p.id, p.name)
}

if (packageIdList.size () > 0) {
    <% Listing the package names that does not start with 'S' %>
```

```

packageIdList->forEach (s:String | not(s.startsWith("S"))) {
  <% Package: %> s
}
<% The package with id='123' : %> packageIdList.get("123")

```

2.9.12 Iterators

Iterators in MOFScript are used primarily for iterating collections of model elements from a source model. The ForEachExp block expression defines a new iterator expression which also has a block of executable expressions.

It works similarly to forAll in OCL or the shorthand iterator expressions from QVT. The example below shows an iterator on a model element collection.

```

self.ownedMembers->forEach (p:uml.Package | p.stereotype = "System") {
  <%
    // the package declration
    package org.mofscript.%> p.name <%
  %>

```

Iterators can also be defined for Collection variables, such as a List. The example below adds three elements to the variable *list1* and prints the contents in the forEach loop.

```

var list1:List
list1.add("E1")
list1.add("E2")
list1.add(4)
list1->forEach(e){
  stdout.println (e)
}

```

String iterators produce loops based on the contents of a String. They may be defined on variables, constants, or literals.

```

var myVar: String = "Jon Oldevik"
myVar->forEach(c )
  stdout.print (c + " ")

"MOFScript"->forEach(c|c.isLowerCase())
  stdout.print(c)

```

Integer iterators produce loops based on the size of the integer. They may be defined on variables, constants, or literals.

```

property aNumber:Integer = 34
aNumber->forEach(n)
  stdout.print(" " + n)

"7"->forEach(n)
  stdout.print(" " + n)

```

2.9.13 If-then-else

If-expressions provide basic functionality for controlling execution based on logical expressions. An if-expression has a condition and a block of statements that are executed if the condition is met. It might have a set of else-if branches and a possible else branch.

```

if (self.name = "Person") {
  <% This is the person type %>
} else if (self.name = "Car")
  <% This is a car type %>
else {
  <% This is neither person nor car type %>
}

```

2.9.14 Invoking rules

Text transformation rules are invoked either directly or as part of expressions.

```

uml.Package::interfacePackages () {
  if (self.getStereotype() = "Service"){
    file (self.name.toLowerCase() + ".wsdl")
    self.wsdlHeader()
    self.wsdlTypes()
    self.ownedMember->forEach(i:uml.Interface) {
      i.wsdMessages()
      i.wsdPortType()
      i.wsdBindings()
      i.wsdService()
    }
    self.wsdlFooter()
  }
}

```

2.9.15 Return results from rules

Text transformation rules may have return results. This is most useful for defining *helper* functions.

```

uml.TypedElement::getType () : String {
  if (self.type.name.equalsIgnoreCase("string"))
    result = "xsd:string"
  else
    result = self.type.name
  }
}

```

2.9.16 Grammar

The MOFScript EBNF grammar is shown below:

```

M2Tmodule      :=      "texttransformation" name "(" (inputModels) ")"
                    extraMetamodels
                    import* use*
                    varOrPropDecl*
                    (m2TRule)+

```

```

inputModels      := "in" <name> ":" <name> ("," inputModels)?
varOrPropDecl   := varDecl | propertyDecl
varDecl          := "var" <name> (":" varType)? ("=" value-
expression)?
propertyDecl    := "property" name (":" varType)? "=" value-
expression
import           := "access" ("library"|"transformation") <name>
"(" <import-uri> ")"
m2TRule         := (context) <name>|"main" parameters
(returnType)? (guard)?
"{" m2TRuleBody "}"
parameters      := "(" type name ("," type name)* ")"
context         := ctxType | "module" "::"
ctxType         := <A metamodel type from an input metamodel>
guard           := <Guards as defined in QVT-Merge proposal>
m2TRuleBody     := (varOrPropDecl)*
(statement)*
statement       := iteratorStatement |
ifStatement |
generalAssignment |
fileStatement |
printStatement |
functionCallStatement |
directPrintStatement |
escapeStatement |
unprotectedBlock

iteratorStatement := collection-ref "->" "forEach"
"("<name> ":" Type ("|" logical-expr)? ")"
blockBody
ifStatement      := "if" "(" logical-expr ")" blockBody
("else" blockBody)?
generalAssignment := ref-name = expression
fileStatement    := ("use"? "file" (ref-name)? ("
(expression |
"name=" expression
("ext=" expression)?
("dir=" expression)?
)")"
printStatement   := (prefix)?("print" | "println") "("
expression ")"
prefix           := (device-reference | "stdout") "."
functionCallStatement := function-reference "(" actual-params ")"
directPrintStatement := (("newline"|"nl") | "tab" | "space")
("(" count ")")?
escapeStatement  := "<% anything %>"
blockBody        := statement | "{" (statement)* "}"
unprotectedBlock := "unprotect" "{" (statement)* "}"
actual-params    := <list of actual params (expression values)>
anything         := <anything except the escape sequence '%>' >
expression       := ... <OCL Expression> ...
varType          := ... "String" | "Boolean" | "Integer" | "Real"
"List" | "Dict" | "Hashtable"
comment          := "--" <any text until eol - QVTcomment> |
"/**" <any text - Java comment> "*/"

```

```
count                :=      <Positive integer number>
```

2.10 DETAILED SEMANTICS

This chapter describes the detailed semantics of the most important concepts in MOFScript.

2.10.1 Files and devices

Output devices provide output capabilities for model to text transformations. Files are considered the most important type of device for text output. Other kinds of devices might be relevant, e.g. SQL DB devices, CVS/SVN devices etc.

A file device declaration has a scope based on the execution of rules. A device declared within a rule is reachable also for rules invoked by the declaring rule. Thus device lifetime is from declaration point until end of the execution of the declaring rule. If rule *x* declares device *fl* and invokes rule *y*, rule *y* can use device *fl*.

A file device is identified by its reference name, or alternatively by the identifying file URI. Two or more references to the same file device has the same semantics as using a single reference.

Output printing that is not prefixed by a device name, is done to the previously declared output device, or the previous device targeted by a *use file* clause. If there is no output device declared, the default is console output (*stdout*). The validity of an output device as default device is from use declaration, through the execution stack until execution end or until a new/different device declaration.

A device goes out of scope when the declaring rule finishes execution.

2.10.2 Print statements

Print statements produce output towards either the current output device or an explicit prefixed output device.

A `print` / `println` statement without prefix will produce output to the current output device. The same will an escaped output statement do.

A prefixed `print/println` statement will produce output on the output device referenced by the prefix device reference, e.g. a file or `stdout`.



3. Compliance points

We suggest two levels of compliance for MOFScript.

All tools which claim full compliance with this submission must implement:

- The full metamodel and lexical syntax, including traceability metamodel
- XMI support for the metamodel

Partial compliance is obtained by implementing the above, less the traceability mechanisms of the metamodel and the lexical syntax.



Part III - Appendices

4. Appendix A Examples

3.1 UML 2 JAVA

The simple example below shows a mapping from a UML 2 metamodel to java class output files.

```

/*
 * Example of Model to Text transformation using the
 * "MOFScript" Language
 * UML 2 Java generation
 */
texttransformation UML2Java (in uml:uml2)

access library variousJavaStuff ("uml2java_include.m2t")

property package_name = "org.sintef.no"
property package_dir = "org/sintef/no/"
property ext = ".java"
property author = "Jon Oldevik"

uml.Model::main () {
  self.ownedMember->forEach(p:uml.Package) {
    p.mapPackage()
  }
}

/*
 * mapPackage
 */

uml.Package::mapPackage () {

```

```

self.ownedMember->forEach(c:uml.Class)
  c.mapClass()

self.ownedMember->forEach(p:uml.Package) {
  p.mapPackage()
}

}

/*
 * mapClass
 */

uml.Class::mapClass() {
  file (package_dir + self.name + ext)
  println (self.classPackage())
  self.standardClassImport ()
  self.standardClassHeaderComment ()

  <%
public class %> self.name <% extends Serializable { %>
  self.classConstructor()
  <%
  /*
   * Attributes
   */
  %>
  self.ownedAttribute->forEach(p : uml.Property | p.association =
null) {
    p.classPrivateAttribute()
  }
  nl(2)
  self.ownedAttribute->forEach(p : uml.Property | p.association !=
null){
    p.classPrivateAssociations()
  }
  nl(2)
  self.ownedAttribute->forEach(p : uml.Property | p.association =
null)
    p.attributeGettersSetters()
  nl(2)

  self.classAssociationMethods()
  self.ownedAttribute->forEach(assoc : uml.Association) {
  }

  <%
} // End of class %> self.name
}

//
// rule ClassPackage
//

```

```

uml.Class::classPackage() : String {
    result = "package " + package_name + ";"
}

//
// class Constructor
//
uml.Class::classConstructor() {
    <%
    // Default constructor
    public %> self.name <% () {
    }%>
    newline
}
//
// classPrivateAttribute
//
uml.Property::classPrivateAttribute () {
    <%
    private %> self.type.name <% __%> self.name.toLower() <% ; %>
}

// classPrivateAssociations
uml.Property::classPrivateAssociations () {
    println ("    private " + self.type.name + "[] _" +
self.name.toLower() + ";")
}

// attributeGetterSetters
uml.Property::attributeGettersSetters () {
    <%
    public %> self.type.name <% get%> self.name.firstToUpper() <% () {
        return __%> self.name.toLower() <%;
    }

    public void set%> self.name.firstToUpper() <%(%> self.type.name <%
__input ) {
        __%> self.name.firstToUpper() <% = __input;
    }
    %>
}

/*
 * Class Association Implementation
 */
uml.Class::classAssociationMethods () {
    self.ownedAttribute->forEach(p : uml.Property | p.association <>
"") {
        p.associationAddMethod()
    }
}

/*
 * Add Method for Association

```

```

    */
    uml.Property::associationAddMethod () {
        <%      public void add %> self.type.name <% ( %> self.type.name <%
obj) {
            _%> self.type.name.toLower() <%.add(obj);
            }%>
            newline(2)
        }

    // standardClassImport
    uml.Class::standardClassImport () {
        <%
import java.util.Vector;
import java.io.Serializable;
        %>
    }

    // standardClassHeaderComment
    uml.Class::standardClassHeaderComment () {
        <%
/**
 *
 * Generated class %> self.name <%
 * @author %> author <%
 *
 */
        %>
    }
}

```

3.2 UML 2 WSDL

```

/*
 *
 * MOFScript UML2 to WSDL Transformation
 *
 */

texttransformation UML2WSDL (in uml:uml2)

access library Wsdllib ("wsdl_library.m2t")

property namespaceBase = "http://mofscript.org/"
property rootdir = "C:/tmp"
property business_service_pack = "BusinessServiceSpec"
property type_namespace = "types"
property hostname = "http://localhost:8080"

//
// main
//
uml.Model::main () {
    self.ownedMember->forEach(p:uml.Package)
        p.interfacePackage()
}

```

```

}

//
// interfacePackage
//
uml.Package::interfacePackage () {
    if (self.name = "Interface Model") {
        self.ownedMember->forEach(p:uml.Package) {
            p.interfacePackages()
        }
    } else {
        stdout.println ("Error in model: 'No Interface model'")
    }
}

//
// interfacePackages
//
uml.Package::interfacePackages () {
    if (self.getStereotype() = business_service_pack){
        file (rootdir + self.name.toLower() + ".wsdl")
        self.wsdlHeader()
        self.wsdlTypes()
        self.ownedMember->forEach(i:uml.Interface) {
            i.wsdlMessages()
            i.wsdlPortType()
            i.wsdlBindings()
            i.wsdlService()
        }
        self.wsdlFooter()
    }
}

/*****
* wsdl Types
*****/
uml.Package::wsdlTypes () {
    property pName = self.name.toLower()
    <%
        <types>
            <xsd:schema targetNamespace="%> namespaceBase + pName <%/>
type_namespace <%/ "
                xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">%>
        self.ownedMember->forEach(c:uml.Class | c.getStereotype()=="Entity") {
            class.wsdlTypeMapping()
        }
    <%
        </xsd:schema>
    </types>
    %>
}

/*

```

```

* wsdl Header
*
*/
uml.Package::wsdlHeader () {
    property pName = self.name.toLower()
    <%<?xml version="1.0"?>
<definitions name="%> self.name <%"
targetNamespace="%> namespaceBase + pName <%.wsdl"
    xmlns:tns="%> namespaceBase + pName <%.wsdl"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:%> type_namespace <%"> namespaceBase + pName <%/>
type_namespace <%/>
    xmlns="http://schemas.xmlsoap.org/wsdl/"

    %>
}

/*
* Wsdl footer
*/
uml.Package::wsdlFooter () {
<%
</definitions>
%>
}

/*
* Wsdl PortType
*/
uml.Interface::wsdlPortType () {
    property portTypeName = self.name + "_PortType"
    <%
    <portType name="%> portTypeName <%">%>
        self.ownedOperation->forEach(o:uml.Operation) {
            o.portTypeOperation()
        }
    <%
    </portType>
    %>
}

/*
* Wsdl Bindings
*/
uml.Interface::wsdlBindings() {
    <%
    <binding name="%> self.name + "_Binding" <%" type="tns:%> self.name
+ "_PortType" <%">
        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>%>
        self.ownedOperation->forEach(o:uml.Operation) {
            o.bindingOperation()
        }
    <%
    </binding>
}

```

```

    %>
}

/*
 * portTypeOperation
 */
uml.Operation::portTypeOperation () {
    <%
        <operation name="%> self.name <%"%>
        if (self.ownedParameter.size() > 0) {
            <% parameterOrder="%> self.getParameterOrder() <%"%>
        }
        <%"%>>
        if (self.ownedParameter.size() > 0) {
            <%
                <input message="tns:%> self.name <_%Request"/>%>
            }
            if (self.returnResult.size() > 0) {
                <%
                    <output message="tns:%> self.name <_%Response"/>%>
                }
            }
        <%
    </operation> %>
}

/****
** Parameter order
*****/
uml.Operation::getParameterOrder () {
    self.ownedParameter->forEach(prm:uml.Parameter) {
        if (prm <> self.ownedParameter.first()) {
            <% %> prm.name
        }
    }
}

/*
 * Binding Operation
 */
uml.Operation::bindingOperation () {
    <%
        <operation name="%> self.name <%">
        <soap:operation soapAction="%> nameSpaceBase +
self.name <%" style="document"/>
        <input>
        <soap:body%>
            if (self.ownedParameter.size() > 0) {
                <% parts="%> self.getParameterOrder() <%"%>
            }
            <% use="literal"/>
        </input>
        <output>
        <soap:body%>
            if (self.returnResult.size() > 0) {
                <% parts="response"%>
            }
        </soap:body%>
    <%
}

```

```

        }
        <% use="literal"/>
    </output>
</operation>
%>
}

/****
** wsdlService
****/
uml.Interface::wsdlService () {
    <%
        <service name="%> self.name <%">
            <port name="%> self.name + "_Port" <%" binding="%> self.name
+ "_Binding" <%">
                <soap:address location="%> hostname + self.name <%">/>
            </port>
        </service>
    %>
}

/*****
** wsdlMessages
*****/
uml.Interface::wsdlMessages () {
    self.ownedOperation->forEach(o:uml.Operation) {
        o.wsdlMessage()
    }
}

/*****
* wsdl Message
*****/
uml.Operation::wsdlMessage () {
    if (self.ownedParameter.size() > 0) {
        <%
            <message name="%> self.name <%"_Request">%>
                self.ownedParameter->forEach(p:uml.Parameter) {
                    <%
                        <part element="%> p.getType() <%" name="%> p.name <%">/> %>
                    }
                <%
            </message>%>
        }
    }
    if (self.returnResult.size() > 0) {
        newline
        <%
            <message name="%> self.name <%"_Response"> %>
                self.returnResult->forEach(p:uml.Parameter) {
                    <%
                        <part element="%> p.getType() <%" name="response"/> %>
                    }
                <%
            </message>%>
        }
    }
}

```

```

    </message>%>
  }
}

//
// getParameterType - returns name of property as String
//

uml.TypedElement::getType () : String {
  result = type_namespace + ":" + self.type.name
  if (self.type.name.equalsIgnoreCase("string"))
    result = "xsd:string"
  if (self.type.name.equalsIgnoreCase("integer"))
    result = "xsd:integer"
  if (self.type.name.equalsIgnoreCase("boolean"))
    result = "xsd:boolean"
}

///
/// wsdl TypeMapping
///
uml.Class::wsdlTypeMapping() {
  <%
    <xsd:element name="%> self.name <%">
      <xsd:complexType>
        <xsd:all>%>
          self.ownedAttribute->forEach(a:uml.Property) {
            <%
              <xsd:element name="%> a.name <%" type="%> a.getType()
            <%" />%>
          }
        <%
          </xsd:all>
          </xsd:complexType>
          </xsd:element>%>
        }
  }

/****
  getStereotype
*****/
uml.Element::getStereotype () : String {
  self.eAnnotations->forEach(annot:uml.EAnnotation | annot.source =
"keywords") {
    annot.details->forEach(d:uml.EStringToStringMapEntry) {
      result = d.key
    }
  }
}
}
}

```

3.3 UML TO HTML

```

/*
 * MOFScript Transformation
 * Simple transformation from UML 2 to HTML
 */

```

```

texttransformation uml2Html (in uml:uml2)

uml.Model::main () {
  file (self.name.toLower() + ".html")
  <%
<html><head><title>%> self.name <% </title> %>
stylesheetLink()
  <%
<body>
  %>
self.ownedMember->forEach(p:uml.Package) {
  p.package2html ();
  }
  <%
</body>
</html>
  %>
}
/*
* package2html
*/
uml.Package::package2html () {
  <%
<h1> Package %> self.name <% </h1>
  <p>Description: %> self.ownedComment <% </p>
  <hr>
  <h2>The classes</h2>
  <ul>

  %>
print(" ");
self.ownedMember->forEach(c:uml.Class){
  c.class2html_link();
  c.class2html_detail();
  }

  <%
  </ul>
  %>
}

/*
* class2html
*/
uml.Class::class2html_link () {
  <%
  <li>
  class <a href="%> self.name <%.html"> %> self.name <%
</a>
  </li>
  %>
}

/*
* uml.Class::class2html_detail

```

```

*/
uml.Class::class2html_detail () {
    file (self.name.toLower() + ".html")
    self.class2htmlHeader ()
    <% Attributes: <br>
        <ul>
    %>
    nl
    self.ownedAttribute->forEach (p:uml.Property) {
        <%
            <li>
                %> p.name <% : %> p.type.name <%
            </li>
        %>
    }
    <% </ul> %>
    self.class2htmlFooter ()
}

// class2htmlHeader
uml.Class::class2htmlHeader () {
    print("<html><head><title> Class " + self.name + "</title>")
    stylesheetLink()
    print("</head><body>\n")
}

// class2htmlFooter
uml.Class::class2htmlFooter () {
    print("</body>")
    print("</html>")
}

// Stylesheetlink
module::stylesheetLink () {
    <% <link rel="stylesheet" type="text/css" href="stylesheet.css"
title="Style"> %>
}

```

References

- [1] Object Management Group. Request for Proposal: MOF Model to Text Transformation RFP, 2004, ad/2004-04-07
- [2] Revised Submission for MOF 2.0 Query/Views/Transformations RFP version 2.1, QVT-Merge Group, OMG document ad/2005-07-01, August 22nd 2005